Dynamic adaptations in ab-initio nuclear physics calculations on multicore computer architectures

Avinash Srinivasa
Masha Sosonkina
Ames Laboratory/DOE
Iowa State University
Ames, IA 50011, USA
{*avinashs,masha*}*@scl.ameslab.gov*

Pieter Maris
James P. Vary
Physics Department
Iowa State University
Ames, IA 50011, USA
{*pmaris,jvary*}*@iastate.edu*

***Abstract*—Computational resource availability often changes during the course of execution of an application. This is especially true in modern multi-user cluster environments where users can run many high-performance applications simultaneously which share resources such as Processing Elements (PEs), I/O, main memory, network. In such a scenario, it would be greatly advantageous to have applications augmented with adaptive capabilities, particularly during runtime. This involves targeting a computationally intensive part of the application and invoking appropriate adaptaions so as to be able to adjust to the dynamically changing system conditions, to prevent drastic performance loss. In this paper, the parallel application MFDn (Many Fermion Dynamics for nuclear structure) used for ab-initio nuclear physics calculations is integrated with a middleware tool for invoking such adaptations. In particular, the multi-threaded Lanczos diagonalization procedure in MFDn is targeted to observe the effect on performance of dynamically changing the number of threads during the iterative process. Performance gains between two to seven times were observed in the presence of competing applications by incorporating these adaptation strategies.**

***Keywords*-MFDn; NICAN; Hybrid MPI/OpenMP; oversubscription**

I. Introduction

The direct solution of the quantum many-body problem transcends several areas of physics and chemistry. Nuclear physics faces the multiple hurdles of a very strong interaction, three-nucleon interactions, and complicated collective motion dynamics. The aim is to solve for the structure of light nuclei addressing all three hurdles simultaneously by direct diagonalization of the nuclear many-body Hamiltonian matrix in a harmonic oscillator basis.

A tool to study nuclear structure is the software package MFDn (Many Fermion Dynamics for nuclear structure) developed by Vary et al [1], [2], [3], [4], [5] at Iowa State University. In MFDn, the nuclear Hamiltonian is evaluated in a large harmonic oscillator basis and diagonalized by iterative techniques to obtain the low-lying eigenvalues and eigenvectors. The eigenvectors are then used to evaluate a suite of experimental quantities to test accuracy and convergence issues.

MFDn has been shown to have good scaling properties using the Message Passing Interface (MPI) [6] on existing supercomputing architectures due to the recent algorithmic improvements that significantly improved its overall performance. In [5], the use of a hybrid MPI/OpenMP approach [7], [8] has been presented to take advantage of the current multi-core supercomputing platforms. Under this approach, MPI and OpenMP [9] are used to communicate among inter-node and intra-node cores, respectively. The number of OpenMP threads that are to be spawned per process is fixed statically at the start of the run and is the same for every MPI process in the execution. This makes sense when running on some of the larger supercomputers or leadership class facilities since here, applications typically get the full use of the node(s) on which they are running.

However, in the case of most interactive cluster environments or cloud computing testbeds, users can run multiple high performance applications simultaneously. As a result, computational resource availability can often change during the run-time of the application. To cope with this, it might be beneficial to have an adaptive algorithm to change the number of threads dynamically based on system information gathered at run-time. However, changing the source code of an application such as MFDn to insert these adaptations is not feasible since it will increase the complexity of the scientific code, which may adversely affect its accuracy and usability. In such a scenario, there is a need for some generic middleware which can monitor the system resources during the execution of the application and invoke appropriate application adaptations. In this work, the middleware tool NICAN [10] is used to monitor the number of threads/processes in the system during the execution of MFDn. Based on this run-time information gathered, the number of threads spawned is changed at regular intervals during the Lanczos diagonalization procedure. (See, e.g., [5] for a description of the Lanczos algorithm.) This particular section of the code is chosen for invoking adaptations due

to its iterative and computationally intensive nature.

The main part of this paper is organized as follows. Section 2 provides an insight into the MFDn code and outlines the possibility of invoking its dynamic adaptations. Section 3 describes the need for integrating middleware with MFDn for invoking adaptations and gives an overview of the NICAN middleware tool used for this purpose. Section 4 presents the MFDn-NICAN integration model and the details of the architecture used for integration. Section 5 describes the experiments carried out along with some performance results and Section 6 includes concluding remarks.

II. Overview of MFDn

The MFDn software is a parallel code for *ab initio* nuclear structure calculations written in Fortran90 and MPI, being actively developed for almost two decades. In the early development of the code [1] and [2], the main focus has been efficient use of memory; significant improvements in its performance have been made over the last 3 years [3], [4], [5], [11], [12] under the US Department of Energy Scientific Discovery through Advanced Computing (SciDAC) Program.

The MFDn code computes a few lowest converged solutions, that is, the eigenvalues (energy levels) and eigenvectors (wave functions), for the many-nucleon Schrödinger equation:

$$H\,|\phi\rangle = E\,|\phi\rangle \ . \tag{1}$$

One key feature of this calculation is the size of the very large sparse Hamiltonian matrix H it can produce. The dimension of the matrix characterizes the size of the many-body basis used to represent a nuclear wave function. In general, the larger the basis set, the higher the accuracy of the energy estimation and other computable quantities one can obtain [13]. MFDn constructs the many-body basis states, the Hamiltonian matrix, and solves for the lowest eigenvalues using the Lanczos algorithm. At the end of a run, it outputs the nuclear wave functions and evaluates selected physical observables, which can be compared to experimental data. The matrix is distributed in a 2-dimensional fashion over the processors, and only the lower triangle is stored and used, because the matrix is symmetric (and real-valued). The Lanczos vectors, needed for re-orthogonalization after every matrix-vector multiplication, are distributed over all the processors. Because of the 2-dimensional distribution of the matrix, MFDn runs on $n(n+1)/2$ processors, where n is the number of diagonal processors.

Since the Lanczos procedure is of particular interest in this work, an overview of the iterative process is provided (see Fig. 1). Each iteration consists of a matrix-vector multiplication, followed by an orthogonalization against all the previous Lanczos vectors (which are also all stored in memory, distributed over all processors). After each matrix-vector multiplication, the resulting vector is accumulated on the n diagonal processors. Next, this vector is distributed to all the processors to do the orthogonalization, and finally the new input vector is distributed to all the processors. After a fixed number of Lanczos iterations (which is an input variable), the lowest eigenvalues and the corresponding wave functions are written to disk from the n diagonal processors.

A. MFDn using Hybrid MPI/OpenMP

Since modern processors are equipped with multiple cores, applications augmented with multi-threading capabilities can become considerably more efficient by making use of the multiple cores and overlapping memory access and communications with computations. To take advantage of the multiple cores, a hybrid MPI/OpenMP approach for MFDn has been presented in [5]. It employs multi-threading using OpenMP directives in the most computationally intensive parts of the code, which are the construction of the Hamiltonian matrix, the Lanczos iterations, and the evaluation of observables.

For the Lanczos iterations, the sparse matrix-vector multiplication is parallelized using an OpenMP *DO* directive to loop over the columns. Due to matrix symmetry, each matrix block is used twice in the multiplication. As a result, each thread has its own private output vectors for the result of the transpose matrix-vector multiplication, which are added inside an OpenMP *CRITICAL* region. The orthogonalization of the output vector against all the previous Lanczos vectors is also parallelized with an OpenMP *DO* directive. A scaling plot showing the improvement in computational performance of MFDn with increase in the number of OpenMP threads is presented in [5].

By using OpenMP directives as described above, parallelism may be achieved by splitting the computation into multiple threads of execution. In the current implementation of the Hybrid MPI/OpenMP approach, the number of threads spawned for the OpenMP regions is determined statically at the start of the MFDn run. However, it might be necessary, during the run, to be able to adapt to the changing system conditions in terms of computational resource availability. In light of this, a strategy which involves changing the number of threads dynamically based on certain information about the state of the system resources at run-time might very well prove to be useful, especially in the presence of any competing applications. This work explores changing the number of OpenMP threads spawned at the beginning of every Lanczos iteration (as illustrated in Fig. 1) using system information gathered at run-time by a middleware engine integrated with MFDn.

III. Using Middleware NICAN with applications

While running parallel and distributed applications, the assumption that the resources are dedicated to running only the current job may be too restrictive. This is especially true in the case of interactive cluster environments or cloud

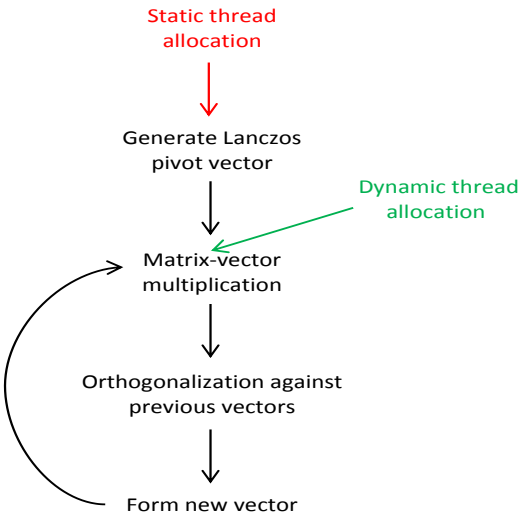


Figure 1. Iterative model for the Lanczos procedure (The Lanczos pivot vector is the initial working vector required for the first MATVEC)

computing testbeds where users can simultaneously run different applications sharing resources, such as Processing Elements (PEs), I/O, main memory, and network. In such cases, system resource availability often changes during the course of execution of the application. This calls for certain run-time adaptations in these applications to be able to adjust to the dynamically changing system conditions. However, it is not desirable to insert these adaptations into the source code of an application, such as MFDn, since this will increase the complexity of the scientific code with adverse affects on its accuracy and usability. The latter is of particular concern since high performance applications are supposed to run on a computational platform by an application scientist who may not be an expert in computer architecture and performance tuning. Hence, there is a need for a generic middleware tool which can monitor the system resources and invoke appropriate run-time adaptations for a large class of applications, so that such a tool may be quickly geared towards a specific application. While leaving its main architecture and system monitoring capabilities intact, the middleware may be augmented by a specific application module [15], thus acting as an interface between the hardware and the application execution. In this work, the middleware tool NICAN [10] developed at Iowa State University is used to serve as an interface with MFDn.

A. NICAN Overview

The main idea of integrating NICAN with an application is to decouple the system-related monitoring and decision making from the execution of the application, while timely invoking application adaptation functions (handlers). The NICAN engine is encapsulated into a separate thread, called Manager, which controls the functional modules and invokes application adaptations. Due to dynamically loadable modules, NICAN is versatile and may have a wide variety of interactions with the system or application. Each module is designated to perform a separable function, such as to determine a system runtime characteristics or to validate machine-dependent parameters. NICAN has a rather general and flexible interaction mechanism, which permits to talk to a variety of application codes. Enhanced with general-use modules, such as CPU monitoring or disk I/O checking, NICAN may not require customized integration with an application. However, to explore application-specific trigger conditions, specific-use NICAN modules may also be needed.

NICAN is mostly used with distributed applications running on many compute nodes of a cluster. The general architecture of integration (Fig. 2) involves a single instance of the Manager on one node, usually on the node on which the rank 0 (root) process is executing, and an instance of the daemon module on each of nodes executing the application. The root node shall henceforth be referred to as the *kickoff* node, with the remaining nodes being referred to as *remote* nodes. The main function of the daemon module is to act as an interface between the Manager and the distributed processes of the application. In some cases, it is also used to pick up system-related information on the remote nodes, which is to be relayed to the Manager to aid the decision making process.

An attractive feature of NICAN is that it does not require substantial coding modifications to the high-performance application with which it is interfaced. In the case of MFDn, only a few changes were made to the source code. Specifically, they include starting up NICAN, tearing it down, and the application specific adaptation handler, such as changing number of threads dynamically based on the information conveyed by NICAN. Resource monitoring, analysis and triggering of the adaptive mechanisms are implemented within the NICAN. Another salient feature of NICAN is to enable or disable its actions with ease and on-demand by the application. This fits very well with the idea of NICAN as a ”black box” from an application scientist’s perspective, abstracting away the details of its functioning.

IV. Integration Model and Adaptation Strategies

A major benefit of integrating NICAN with an application is to separate the system-based monitoring from the invocation of adaptations which are application related. The MFDn-NICAN integration may accomplish the goals described in this work by NICAN monitoring the workload on a core and deciding the number of threads to be spawned by MFDn at particular iterations of the Lanczos process. By collecting the information on the number of running threads/processes resident in the system, a decision may be made to change the thread count for the next iteration

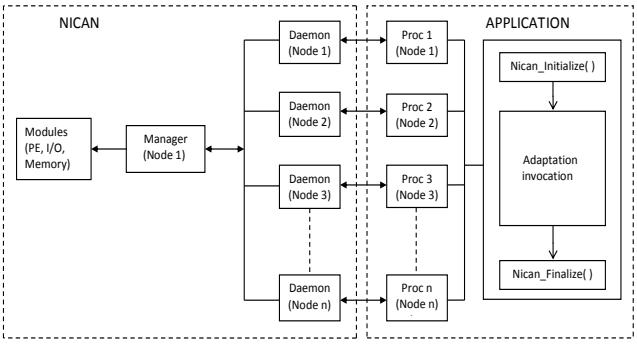


Figure 2. General architecture for integration of NICAN with a distributed application

in order to avoid the oversubscription of a core. The core oversubscription occurs when more than one thread is running on the core processing element and has been shown to be detrimental for the application performance [14]. It is clear that the oversubscription monitoring and avoidance is a dynamic process which may not be dealt with effectively using static tuning and configuration. Thus, dynamic resource monitoring and interfacing with applications as provided by NICAN is fully exploited in this work.

A. Oversubscription and context switching

Processors equipped with multiple cores have become ubiquitous in modern high-performance computers. The number of cores is growing higher in order to keep up with Moore's law, further aggravating the "Memory Wall" which is caused by the inability of memory access to keep up with the speed of processing. In such a scenario, to prevent cores from idling, the use of multi-threading can increase the efficiency of applications by masking memory accesses and communications with computations. Having multiple threads of execution in applications enables to extract thread-level as well as instruction-level parallelism on modern multi-core architectures.

The concept of multi-threading brings into light the idea of context switching. A context switch means storing and restoring the processor state to resume execution from the point where the switch occurred. With regard to threads, it means switching the flow of execution among the different threads which execute on a single PE core sharing the same functional units and execution pipeline, in addition to resources such as caches and TLB (Translation Lookaside Buffer). The intervals between which context switching occurs are determined by the operating system scheduler which usually gives a time slice to each thread/process before preemption and control switching over to another thread/process. Context switches can be detrimental to the performance of a multi-threaded application due to the scheduler overhead of switching among the threads which execute on the same core. This process of switched execution of threads is known as *Simultaneous Multi-Threading (SMT)* in operating system parlance.

However, most modern operating systems are well equipped to handle multi-threading depending on the number of cores available on a processor. The threads are usually distributed over the all the cores by the scheduler. This ensures that they truly execute in parallel, since each PE core has its own functional units and execution pipeline. Thus, to exploit the benefits of multi-threading, it is best to have the number of threads equal to the number of cores. To have more threads than the number of cores is commonly known as *oversubscription* because of the overhead incurred in context switching among the threads executing on the same core.

B. Architecture of integration

In MFDn, as per the hybrid MPI/OpenMP approach presented in [5], MPI is used for distributed memory communication among the nodes with OpenMP being used to spawn multiple threads within a node. The aim is to increase efficiency by taking advantage of the multiple cores and shared memory structure on the node. Under the hybrid MPI/OpenMP approach on a majority of architectures, we would want to run MFDn with one MPI process per node and the number of threads per process equal to the number of cores on a node to prevent any cores from idling. However, in realistic situations, there is a possibility of applications, e.g., run by other users on the same node(s), competing for the PE resources. This causes context switching to occur which can be detrimental to the performance of MFDn. This is especially true when the competing threads/processes are compute-cycle intensive, as is commonly the case with large scale scientific applications. Hence, there is a need to develop strategies for detecting possible oversubscription during run-time and invoking appropriate adaptations in MFDn to be able to adjust to the changing system conditions. In this section, the architecture employed for the MFDn-NICAN integration is described along with the system monitoring and decision making strategies used for the invocation of adaptations in MFDn. These strategies are sufficiently general to be employed with other applications.

In Section 3, the general architecture for the integration of NICAN with an application was explained along with the main components of the NICAN engine. The same architecture is employed for the integration with MFDn with the modules used for system monitoring geared towards the need to change the number of threads spawned by MFDn at run-time. Fig. 3 depicts the architecture used for the integration at the Lanczos stage of MFDn.

The MFDn-NICAN integration model includes a PE load module which monitors the number of running threads/processes in the system during the run-time of MFDn. The main function of this module is to detect if there is oversubscription on any of the cores due to the presence

of competing applications. This module is loaded by the NICAN Manager thread, which is started on the *kickoff* node and which interfaces with MFDn via the daemon module. A daemon instance is started on every node which executes the application. Besides being the Manager's contact point for all the NICAN-integrated applications running on the node, the daemon module also picks up information regarding the number of threads to be relayed to the Manager.

The Manager uses the thread information obtained from the PE and daemon modules to make a decision regarding the number of threads to be spawned by the Lanczos iterative algorithm at the beginning of an iteration. It then invokes the appropriate adaptations via the adaptation function. A simple algorithm is employed for the decision making process. Define (1) the total number of running threads in the system as Θ_s, (2) the number of MFDn threads spawned Θ_a, (3) the number of competing application threads Θ_c and (4) the number of PE cores per node K, the number of MFDn threads that should be spawned for the ith iteration for a particular node is calculated using Algorithm 1.

Algorithm 1 Number of MFDn threads to be spawned at the ith Lanczos iteration

$\Theta_c(i) = \Theta_s(i) - \Theta_a(i-1) - 1$
if $\Theta_c(i) \geq K$ **then**
  $\Theta_a(i) = 1$
else
  $\Theta_a(i) = K - \Theta_c(i)$
end if

Θ_s is found as a result of the monitoring process. K is a known constant for a node. In the first line of Algorithm 1, the current number of MFDn threads along with the thread which does the actual monitoring are subtracted from Θ_s. The NICAN thread is very lightweight and hence does not cause any performance penalty in MFDn due to context switching.

With regard to implementation, information retrieved from the `/proc/loadavg` file is used by the NICAN modules to find the number of running threads/processes in the system during the run-time of MFDn. The number of threads as returned by NICAN after the decision making process is spawned for a particular iteration by MFDn by making use of the OpenMP function `omp_set_num_threads(numthreads)`. Here, `numthreads` refers to the number of threads to be spawned for a subsequent OpenMP region. Communication between NICAN and the distributed processes of MFDn is established using the TCP/IP socket library.

V. Experimentation and evaluation

In this section, the experiments conducted with the MFDn-NICAN integrated model are presented along with some results that were observed in terms of improvement in performance with the inclusion of dynamic adaptations. In these experiments, the performance of the MFDn-NICAN code is tested in comparison with MFDn executing in a cluster environment. The aim is to observe the impact of changing the number of OpenMP threads spawned dynamically as opposed to running with a fixed number of threads. The NICAN middleware tool is used to gather information about the system, in particular, the number of running threads/processes sharing the PE resources on each node. Based on this information, a decision is taken by the tool regarding the number of threads to be spawned which is then relayed to MFDn at regular intervals to invoke the appropriate adaptations.

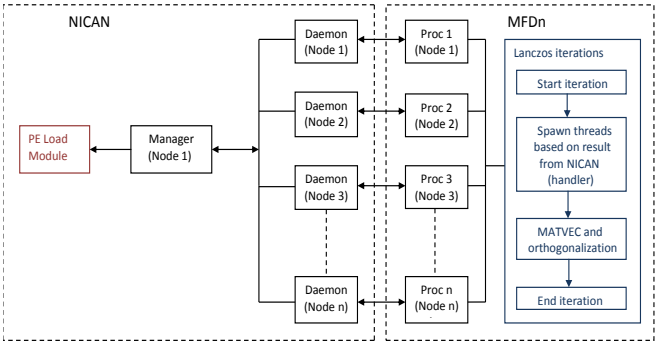


Figure 3. Architecture of the MFDn-NICAN integration

The testbed used for the experiments was the "Dynamo" cluster consisting of 34 SMP compute nodes, each having two quad-core 3.0 GHz Intel Xeon E5450 processor chips and 16 GB of RAM, i.e., equipped with a total of 8 cores per node and 272 cores overall. The nodes are connected with both Gigabit Ethernet and DDR Infiniband. In these experiments, both MFDn and MFDn-NICAN are run for ^{12}C nucleus using six MPI processes, one on each node. Furthermore, each process spawns eight OpenMP threads, which is specified at the start of run, thus ensuring that none of the cores on a node are left idle. For these experiments, multi-threaded regions are defined only during the Lanczos iterations with the rest of the code being single threaded. With MFDn-NICAN, the number of threads is subject to change during the course of the run depending on the PE resources available, while it is held constant throughout the run in the case of pure MFDn. The aims are (1) to consider the penalty incurred due to context switching in the presence of any application which competes for the same PE resources and (2) to show the usefulness of integrating NICAN with MFDn in coping with such a situation.

The tests were carried out for two problem sizes, $N_{max} = 2$ and $N_{max} = 4$ for the ^{12}C nucleus. The problem size is characterized by the dimension of the Hamiltonian matrix which is 17,725 for $N_{max} = 2$ with 1,697,935 non-zero

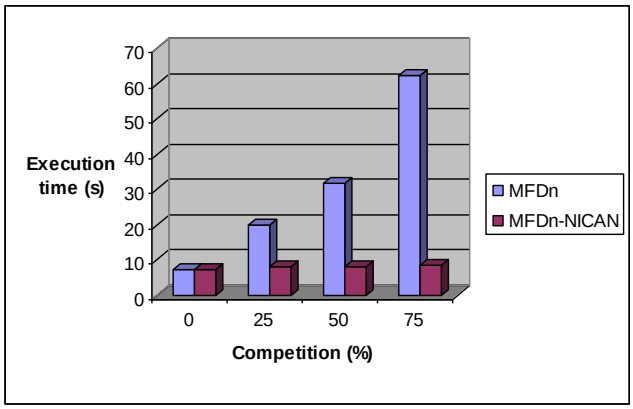


Figure 4. Comparison of execution times of MFDn and MFDn-NICAN for ^{12}C nucleus with $N_{max} = 2$.

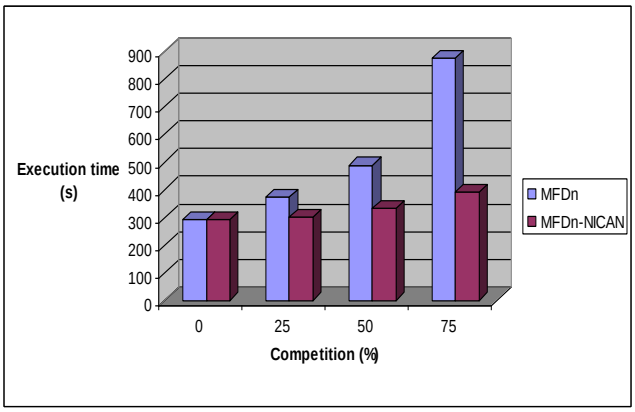


Figure 5. Comparison of execution times of MFDn and MFDn-NICAN for ^{12}C nucleus with $N_{max} = 4$.

elements and 1,118,926 for $N_{max} = 4$ with 279,405,126 non-zero elements. In general, as the N_{max} value increases, so does the size of the Hamiltonian matrix yielding more computationally-intensive and more accurate calculations. The bar graphs in Fig. 4 and 5 depict performance results that were obtained for both MFDn and MFDn-NICAN for these two problem sizes with varying degrees of competition. The competition is defined as the percentage of the total (8) cores per node occupied with other high-performance applications. For the purpose of competition, the quantum chemistry software GAMESS [16] was used in the direct configuration which has been shown to be compute-cycle intensive. GAMESS processes were introduced on the nodes during the run-time of MFDn and their impact on the performance for both pure MFDn and MFDn-NICAN was observed.

From the two graphs (Fig. 4 and 5), it can be clearly seen that under increasing competition, the performance

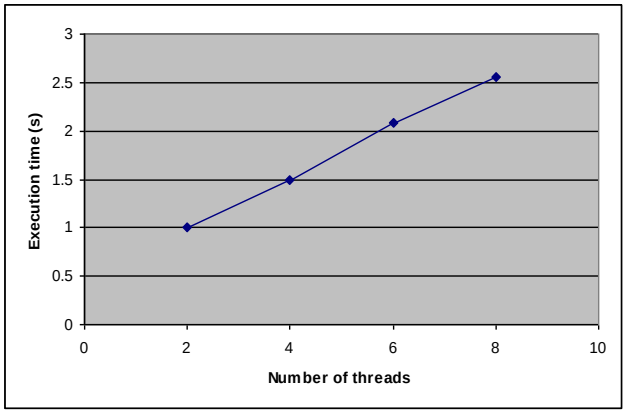


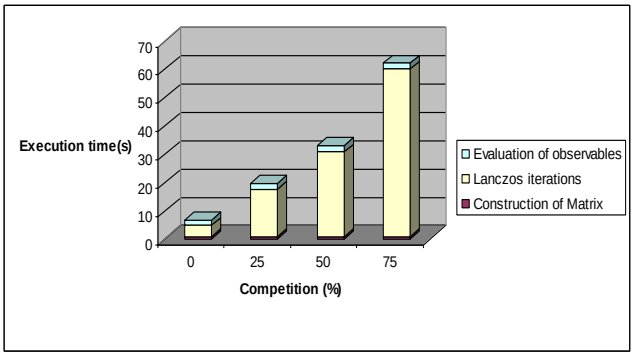
Figure 6. Scaling of the Lanczos iterative phase with number of threads per MPI process.

of MFDn reduces considerably due to context switches happening on more cores. On the other hand, with MFDn-NICAN, the performance is much better, being almost equal to that of MFDn with full resource availability (i.e., with no competition) for the lower problem size $N_{max} = 2$. For $N_{max} = 4$, however, the performance of the adaptive algorithm does not reach the peak performance since with competition from other applications, it is forced to run on fewer threads than the maximum possible. This hinders the performance for larger problem sizes, as in the case of $N_{max} = 4$ here, that require full power of the node PE resources. Such a tradeoff is acceptable, however, since the performance penalty incurred due to context switching between MFDn and competing applications leads to a much slower execution. As is evident from the graphs, the penalty increases with the increase in competition.

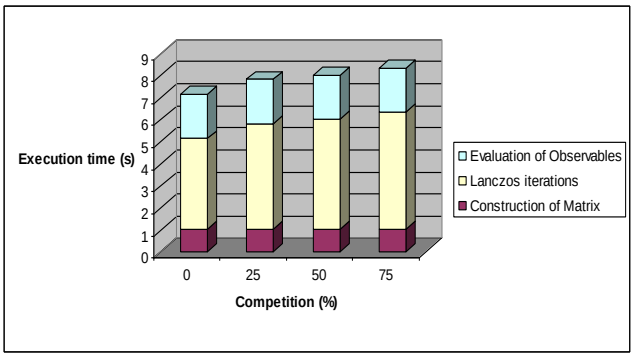
Fig. 6 depicts the speedup obtained for the multi-threaded Lanczos iterative procedure with the number of threads for the larger problem size $N_{max} = 4$. It can be seen from the graph that the scaling is not entirely ideal in moving from 2 to 8 threads. This can be attributed to the fact that there is a significant amount of MPI communication in this phase of the code which proves to be a performance bottleneck. Multithreading speeds up the local computations on each node, but not the MPI communication between the nodes. The serialization due to the OpenMP *CRITICAL* region as explained in Section 2 also contributes to the imperfect scaling.

Fig. 7 and 8 illustrate the performance of various sections of the MFDn code, again with different degrees of competition for both the non-adaptive and adaptive algorithms. This serves to determine which section is incurring the most performance penalty due to the context switching. This is again shown for the same two problem sizes i.e., $N_{max} = 2$ and $N_{max} = 4$.

It can be seen that the multithreaded Lanczos iterative

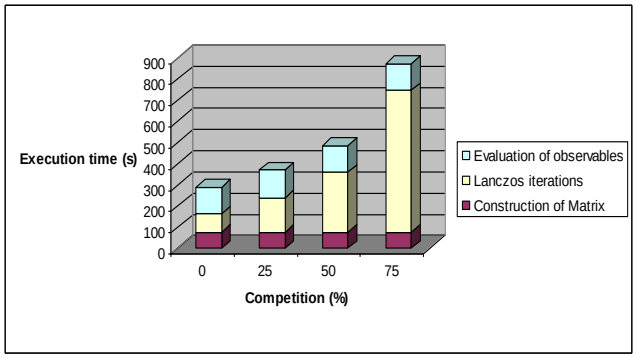


(a) MFDn

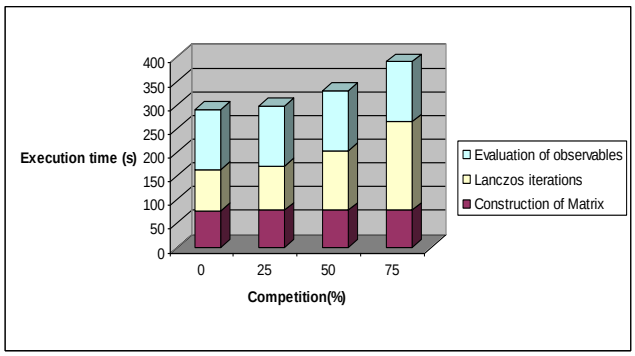


(b) MFDn-NICAN

Figure 7. Execution times of different sections of the code with different degrees of competition for ^{12}C nucleus with $N_{max} = 2$.



(a) MFDn



(b) MFDn-NICAN

Figure 8. Execution times of different sections of the code with different degrees of competition for ^{12}C nucleus with $N_{max} = 4$.

procedure incurs a higher penalty as the degree of competition increases. Thus, it bears the primary responsibility for the performance decrease whereas the other sections, namely, the construction of the Hamiltonian matrix and the evaluation of observables, which are single-threaded, retain the same performance even in the presence of competition. This is not surprising since the multiple threads in the Lanczos procedure are spread across all the cores and undergo context switching in the presence of competition on any of the cores while the calculations in the other two sections are undisturbed since they enjoy a dedicated core. (Note that the maximum competition is 75% meaning that MFDn has always a sole use of at least two cores, on which it performs the construction of Hamiltonian and the observable calculation.)

These experiments indicate the usefulness of including adaptive capabilities in MFDn by dynamically changing the number of threads to deal with the problems of oversubscription and context switching. The advantage of using a middleware for this purpose, as explained in the earlier sections, is to decouple the system-related monitoring and decision making from the execution of the application without incurring much performance and interfacing overhead for adaptation. The performance gains obtained as a result validate this adaptive approach for MFDn. Since a generic middleware tool is employed to implement this approach, the strategies presented here may be extended to other multi-threaded distributed high performance applications.

VI. Conclusions and Future Work

The main contribution of this work is the inclusion of dynamic adaptations in MFDn, a large scale parallel code used for ab-initio nuclear physics calculations, by integrating it with the middleware tool NICAN. The tool monitors the system resources during the run-time of MFDn and makes a decision on the number of threads to be spawned by the multi-threaded Lanczos procedure during the iterative process. As a result, MFDn self-adapts to the dynamically changing system conditions, such as PE resource availability. In this paper, PE resource availability has been tested using competing applications that might execute simultaneously in the non-disjoint subsets of nodes in a cluster environment.

The integration of MFDn with the middleware tool NICAN proved to be useful for facilitating MFDn adaptations in a non-intrusive manner. Adaptation decision-making strategies may be implemented in NICAN as application-specific or general-purpose modules. Such strategies have been tested with for two different problem sizes for MFDn by introducing various degrees of competition for the PE

cores of the nodes executing MFDn. The results obtained validated concern about the multi-threaded application performance degradation in the presence of applications competing for PE resources (cores) even when other resources, such as memory, suffice. The proposed adaptation strategies brought about significant performance gains: more than twofold improvement for very large problem sizes and surpassing a seven-fold improvement for the problem sizes that do not place excessive demands on the single-node PE resources. The adaptation of the number of threads available to the application to eliminate the context switches by the operating system is general. Thus, it may be used by a wide class of applications with a computationally-intensive iterative calculation.

The system monitoring process employed here is largely local in nature, in the sense that each node has its own PE which may be monitored independently. Hence, the proposed adaptation strategy suits well massively parallel architectures and may be employed in conjunction with global performance enhancing techniques thereby creating a multi-level adaptation. The exploration of hierarchical adaptations is left as future work.

ACKNOWLEDGMENT

This work was supported in part by Iowa State University under the contract DE-AC02-07CH11358 with the U.S. Department of Energy, by the U.S. Department of Energy under the grants DE-FC02-09ER41582 (UNEDF SciDAC-2) and DE-FG02-87ER40371 (Division of Nuclear Physics), by the Director, Office of Science, Division of Mathematical, Information, and Computational Sciences of the U.S. Department of Energy under contract number DE-AC02-05CH11231, and in part by the National Science Foundation grant NSF/OCI – 0904782.

REFERENCES

[1] J. P. Vary, *The Many-Fermion Dynamics Shell-Model Code*. Iowa State University (unpublished) (1992)

[2] J. P. Vary, D. C. Zheng, *The Many-Fermion Dynamics Shell-Model Code*. (unpublished) (1994)

[3] P. Sternberg, E.G. Ng, C. Yang, P. Maris, J.P. Vary, M. Sosonkina, H.V. Le, *Accelerating configuration interaction calculations for nuclear structure*. In Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, Conference on High Performance Networking and Computing, Austin, Texas, November 15 - 21, 2008, pp. 112. IEEE Press, Piscataway (2008), http://doi.acm.org/10.1145/1413370.1413386

[4] M. Sosonkina, A. Sharda, A. Negoita, J.P. Vary, *Integration of Ab Initio Nuclear Physics Calculations with Optimization Techniques*. In Bubak, M., van Albada, G.D., Dongarra, J., Sloot, P.M.A. (eds.) ICCS 2008, Part I. LNCS, vol. 5101, pp. 833842. Springer, Heidelberg (2008)

[5] P. Maris, M. Sosonkina, J.P. Vary, E.G. Ng, C. Yang, *Scaling of ab-initio nuclear physics calculations on multicore computer architectures*. In Procedia Computer Science, Volume 1, Issue 1, May 2010, Pages 97-106, ICCS 2010

[6] M. P. I. Forum, *MPI: A message-passing interface standard*. (1994)

[7] E. L. Lusk, A. Chan, *Early experiments with the OpenMP/MPI hybrid programming model*. In R. Eigenmann, B. R. de Supinski (Eds.), OpenMP in a New Era of Parallelism, 4th International Workshop, IWOMP 2008, West Lafayette, IN, USA, May 12-14, 2008, Proceedings, 2008, pp. 3647

[8] R. Rabenseifner, G. Hager, G. Jost, *Hybrid MPI/OpenMP parallel programming on clusters of multi-core smp nodes*. In D. E. Baz, F. Spies, T. Gross (Eds.), Proceedings of the 17th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, PDP 2009, Weimar, Germany, 18-20 Febuary 200, 2009, pp. 427436

[9] L. Dagum, R. Menon, *OpenMP: An industry-standard api for shared-memory programming*. Computing in Science and Engineering 5 (1998) 4655. doi:http://doi.ieeecomputersociety.org/10.1109/99.660313.

[10] M. Sosonkina, *Adapting Distributed Scientific Applications to Run-time Network Conditions*. In J. Dongarra, K. Madsen and Jerzy Wasniewski, editors, Applied Parallel Computing, State of the Art in Scientific Computing, 7th International Workshop, PARA 2004, Revised Selected Papers, volume 3732 of Lecture Notes in Computer Science, pages 745755. Springer, 2006

[11] J. P. Vary, P. Maris, E. Ng, C. Yang, M. Sosonkina, *Ab initio nuclear structure - the large sparse matrix eigenvalue problem*. J. Phys. Conf. Ser. 180 (2009) 012083. arXiv:0907.0209, doi:10.1088/1742-6596/180/1/012083

[12] N. Laghave, M. Sosonkina, P. Maris, J. P. Vary, *Benefits of parallel i/o in ab initio nuclear physics calculations*. In Computational Science - ICCS 2009, 9th International Conference, Baton Rouge, LA, USA, May 25-27, 2009, Proceedings, Part I, 2009, pp. 8493.

[13] P. Maris, J. P. Vary, A. M. Shirokov, *Ab initio no-core full configuration calculations of light nuclei*. Phys. Rev. C 79, 014308 (2009) arXiv:0808.3420 [nucl-th]

[14] P. Apparao, R. Iyer, D. Newell, *Towards Modeling and Analysis of Consolidated CMP Servers*. Workshop on the Design, Analysis, and Simulation of Chip Multi-Processors (dasCMP), 2007

[15] N. Ustemirov, M. Sosonkina, M. S. Gordon, M. W. Schmidt, *Dynamic Algorithm Selection in Parallel GAMESS Calculations*. International Conference Workshops on Parallel Processing, (ICPPW'06)

[16] M. W. Schmidt, K. K. Baldridge, J. A. Boatz, S. T. Elbert, M. S. Gordon, J. H. Jensen, S. Koseki, N. Matsunaga, K. A. Nguyen, S. Su, T. L. Windus, M. Dupuis, J. A. Montgomery Jr, *General atomic and molecular electronic structure system*. Journal of Computational Chemistry, pages 1347-1363, November 1993